Establishing Qualitative Software Metrics in Department of the Navy Programs

Chris Johnson
Ritesh Patel
Deyanira Radcliffe
Paul Lee
John Nguyen
SPAWAR System Center Pacific
San Diego, California USA

***Abstract*—Software Development is an infant engineering discipline compared with other engineering areas. The ability of programs, Program Managers, and Lead Software Engineers to effectively measure how a program is doing is increasingly difficult based on shifting requirements, resource constraints, familiarity of the development domain, etc.**

The Department of the Navy is dedicated to provide the highest quality software to its users. In doing, there is a need for a formalized set of Software Quality Metrics. The goal of this paper is to establish the validity of those necessary Quality metrics. In our approach we collected the data of over a dozen programs from previous tests, analyzed current states of the software, derived formulas via weighting to provide necessary results, investigated tool sets to provide the necessary variable data for our formulas and tested the formulas for validity. Keywords: metrics; software; quality

I. Purpose

Space and Naval Warfare Systems Center Pacific (SSC Pacific) seeks to establish and provide a set of software quality metrics, measured from common static code analysis tools which the Department of the Navy can use to measure quality. These metrics provide quality and maturity data through all stages of software development to further ensure that the software delivered meets government-specific requirements. Carefully chosen metrics can direct attention to problems, providing diagnostic value and influence developers' behavior, and offset post-delivery maintenance costs.

C. Johnson is with SPAWAR Systems Center Pacific, San Diego, CA. 92152, USA; email: chris.e.johnson@navy.mil
R. Patel is with SPAWAR Systems Center Pacific, San Diego, CA. 92152, USA; email: ritesh.patel@navy.mil
D. Radcliffe is with SPAWAR Systems Center Pacific, San Diego, CA. 92152, USA; email: deyanira.radcliffe@navy.mil
P. Lee is with SPAWAR Systems Center Pacific, San Diego, CA. 92152, USA; email: paul.h.lee2@navy.mil
J. Nguyen is with SPAWAR Systems Center Pacific, San Diego, CA. 92152, USA; email: jnguyen@spawar.navy.mil

II. Software Quality Characteristics

Software developers can use common static code analysis tools to obtain various measurable metrics of various categories from every software component. This document identifies software qualities and their indicators that affect DoD 5000.02 program areas, primarily cost, schedule, and risk. For software quality measures the following abilities associated with any software are considered.

- Reusability
- Portability
- Maintainability
- Security
- Extensibility
- Reliability
- Testability
- Scalability

III. Software Quality Test Assertions

The test looked at over forty software applications, from over a dozen different developers, including government and contractor. In order to achieve comparative measurements, applications with less than four hundred thousand source lines of code were selected. All the applications selected had previously been tested, and in many cases operationally fielded. From operational use, empirical data was available to support a familiarity of the actual quality of the software prior to the test. This data enabled us to refine the formulas through test and analysis, formula refinement, and retest, to adjust the formulas and weighting and provide us the expected results.

Next, we selected fifteen test tools to support various measurements and program languages platforms. Once the tools were determined, software applications were tested to generate the necessary quality attribute data. The data provided from the tools allowed for creation of the formulas and weighting necessary, to achieve overall qualitative measurement of software.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
29 OCT 2015	**N/A**	**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Establishing Qualitative Software Metrics in Department of the Navy Programs	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
Chris E. Johnson Ritesh Patel Deyanira Radcliffe Paul Lee John Nguyen	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **SPAWAR Systems Center Pacific**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

AbstractâSoftware Development is an infant engineering discipline compared with other engineering areas. The ability of programs, Program Managers, and Lead Software Engineers to effectively measure how a program is doing is increasingly difficult based on shifting requirements, resource constraints, familiarity of the development domain, etc. The Department of the Navy is dedicated to provide the highest quality software to its users. In doing, there is a need for a formalized set of Software Quality Metrics. The goal of this paper is to establish the validity of those necessary Quality metrics. In our approach we collected the data of over a dozen programs from previous tests, analyzed current states of the software, derived formulas via weighting to provide necessary results, investigated tool sets to provide the necessary variable data for our formulas and tested the formulas for validity.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**SAR**	**12**	

IV. Software Quality Measurment

Table I defines a matrix for determining a score for reusability. It is as an example for the other abilities being measured in this document. The columns in the tables represent the software attributes our research proved to be the most relevant to determine quality for software we acquire. The rows in the table define a range of values to score the software. The project team selected these attributes based on various documents, studies, academic research, industry findings, and empirical data of locally developed programs as listed in References [1-33].

To achieve the overall score of a particular ability, we selected the combined measures from the table. The corresponding grade 1-5 was selected for each attribute. The grade number for each attribute is then multiplied by the corresponding weighting in the formula. The numbers for each attribute were then added to arrive at a final score. That score, using the same overall grade as the individual attributes, was used to determine the overall quality.

A. Reusability

Software abstractness drives reusability. Abstract software can be inherited, which allows for increased reuse. In addition to abstract software modularity improves software reuse since smaller more abstract components can be reused and put together like LEGO® blocks to create new functionality.

Reusability must consider input and output parameters in order to be successfully implemented. Not only must reusability metric involve a wide variety of input parameters, it must define the interrelationships among the parameters and their relative importance [1]. The formula provides heavier weighting on abstractness (0.5) over the other contributing variables in the formula. Based on our research abstract software is composed of modular components that are more general making adoption by other classes easier. Our formula utilized the Chidamber and Kemerer OO Metrics in addition to elements as defined in Ref [2]. Using the same set of core metrics [2], modularity provides the next higher weighting, (0.3) which accounts for the sizes of the modules and actual number of modules being used for the application. For this measure, smaller module sizes with more modules are preferred, and provide the associated formula weighting. Studies indicate that complex modules cost the most to develop and have the highest rates of failure [1]. Complexity and architecture provide the final attribute measures, and are weighted identical based on once module sizes are reduced, best engineering process would dictate decreasing the complexity of the modules and thus help in achieving the modularity values desired.

Related Metrics:

- Modularity
  - Number of module score
  - Module size score
- Abstractness
  - Weighted Methods per Class (WMC)
  - Number of Children per Class (NOC)
- Complexity
  - Cyclomatic Complexity
  - Coupling
- Open Architecture Assessment
  - Use open architecture = 1 if not 0

TABLE I. Reusability Score Matrix

Grade		Modules			Abstractness		Complexity
		Number of Modules	Coupling	Size	WMC	NOC	Cyclomatic Complexity
1	Very Good	>20	<50%	0-200	1-2	25% >	<3
2	Good	15-20	51-60%	201-300	3-5	20-25%	3-6
3	Fair	11-15	61-75%	301-400	6-9	10-20%	7-10
4	Needs Improvement	5-10	76-90%	401-500	10-14	5-10%	11-14
5	Poor	1-5	>90%	>500	>14	<5%	15-20

Calculations:

Modularity Calculation:
$Mo = (Mn \times .5) + (Ms \times .5)$

Abstractness Calculation:
$Ab = (W \times 0.5) + (N \times 0.5)$

Complexity Calculation:
$Co = (Vg \times 0.5) + (Cp \times 0.5)$

Reusability Calculation
$Re = Mo(0.3) + Ab(0.5) + Co(0.1) + Oa(0.1)$

Re = Reusability
Mo = Modularity
Mn = Number of Modules
Ms = Module Size
Ab = Abstractness
Co = Complexity
Vg = Cyclomatic Complexity
Cp = Coupling
Oa = Open Architecture Assessment
W = Weighted Methods per Class
N = Number of Children

B. Portability

Software portability entails the ability and effort required to produce a runnable application based on existing source code for a new environment. Portability is the ability to move software among different runtime platforms without having to rewrite it partly or fully [3]. Software portability depends on the language used, the libraries, the dependency on native

system calls, and the assumptions about the underlying hardware, including display, storage space, memory availability and permissions. Measuring portability is not a simple task. Product metrics may measure size, performance, complexity (density of branches, variables, etc., procedure or module dependencies), or occasionally more abstruse attributes like reliability or maintainability [4]. The portability metric is useful, but it is critical to first review the software architecture to determine the availability of dependent libraries as well as hardware assumptions. For portability there are two important factors to consider prior to determine the degree of portability. First, does the cost of porting provide the necessary incentive to port versus conduct new development. And secondly, does the software to be ported align to the questions formed below in a pre-gate evaluation of overall system portability. If it is determined that portability is cost prohibitive or determined that the target environment cannot support the software to be ported, then calculating the degree of portability and determining a portability score is unnecessary. The key features for software portability are as derived from Ref. [4, 5]:

- Use of popular high-level language (not assembly language, or platform-specific language)
- Keeping platform-specific code in modules separate from the cross-platform code and building applications on a platform-abstraction layer.
- Use of standardized and widely available APIs (e.g., OpenGL™, X) and cross-platform network APIs, protocols, and data representations (e.g., XML, JSON Studio™, CORBA); pay attention to byte-ordering, structure-packing, and native character set issues.
- Use of cross-platform libraries and open source libraries that have multiplatform support.
- Use of a cross-platform virtual machine or interpreter (e.g., Java™, Smalltalk, Python™).

Related Metrics:

- Programming languages
  - Software is not portable if it is written using platform specific language or language that is not supported on the targeted platform
  - Java™, C, C++, Python™ = 2, other high level language = 1
- Architecture assessment
  - Interview the system architect or lead programmer and check off the architecture features for score
- Modularity
  - Number of modules
  - Module size score
- Complexity
  - Cyclomatic complexity
  - Coupling

TABLE II. PORTABILITY SCORE MATRIX

Grade		Programming Languages	Open Architecture Score	Modules			Complexity
				Number of Modules	Coupling	Size	Cyclomatic Complexity
1	Very Good	Java, C, C++, Python, Perl	=1	>20	<50%	0-200	<3
2	Good		1-2	15-20	51-60%	201-300	3-6
3	Fair	Other High Level Language	2-3	11-15	61-75%	301-400	7-10
4	Needs Improvement		3-4	5-10	76-90%	401-500	11-14
5	Poor	Platform Specific Language	4-5	1-5	>90%	>500	15-20

A careful examination of all hardware dependencies is an important first step as hardware dependencies present the biggest challenge in portability. Whether it is a smart card, a display device, a storage device, or some specialized hardware, when support for the hardware does not exist on the targeted platform, the project is considered not portable and the grade for the portability category is Poor for all categories.

Portability Pass/Fail questions:

- Is there any critical hardware dependency where support does not exist on the targeted platform?
- Is there platform specific language in the software?

After passing the portability questions, the architecture score for portability can be determined by reviewing the software architecture or interviewing the system designer. The following questions should be answered:

- Does the project use a cross-platform virtual machine or primarily use interpreted language (e.g., Java™, Smalltalk, and Python™)?

  [Yes = Very good portability, Grade = 1] (100% of final grade)
- If a cross-platform virtual machine or interpreter is not used. How well is the platform-specific code separated from the cross-platform code?

  [Estimate using the very good, good, fair, needs improvement and poor grades.] (33% of final grade)
- Use standardized and widely available APIs (e.g., OpenGL™, X) and cross-platform network APIs, protocols, and data representations (e.g., XML, JSON Studio™, CORBA). Pay attention to byte-ordering, structure-packing, and native character set issues.

  Estimate using the very good, good, fair, needs improvement and poor grades. (33% of final grade)
- Use cross-platform libraries and open source libraries that have multiplatform support.

Yes = Very good portability, grade = 1

No = Poor portability, Grade =5 (33% of final grade)

Programming languages score:

- Score assignment for Java™, C, C++, or Python™ is "Very Good". Use "Fair" for other high level language.

Calculations for portability are based on architecture assessment, modularity, complexity and programming languges. The weighting associated with these attributes determined by our research and showed that modularity and complexity were both weighted to account for 60% of Portability score. These values are based on existing research identifying more modular and less complex software as being integral to portability[4]. In addition to modularity and complexity our research indicated that we also need to factor targeted architecture and types of programming languages that make up the software system. These two items were weighted equally to provide remaning 40% of the total portability score.

Calculations:

Modularity Calculation:

$Mo = (Mn \times 0.5) + (Ms \times 0.5)$

Complexity Calculation:

$Co = (Vg \times 0.5) + (Cp \times 0.5)$

Portability Calculation:

$P = Oa(0.2) + Mo(0.3) + Pl(0.2) + Co(0.3)$

P = Portability
Oa = Open Architecture Assessment
Mo = Modularity
Mn = Number of Modules
Ms = Module Size
Cp = Coupling
Pl = Programming Languages
Co = Complexity
Vg = Cyclomatic Complexity

C. Maintainability

As technology, security risks, and hardware requirements increase, software must evolve to continue to function optimally. The need to maintain the software becomes a critical expenditure to ensure regular updates and revisions that correct any issues, improve efficiency and maintain security.

The programmers' opinion regarding the level of maintainability usually varies. However, generally, according to their opinion a program with a high level of maintainability should consist of modules with strong cohesion and loose coupling, readable, clear, simple, well–structured and sufficiently commented code, having a strictly concurrent style and well–conceived terminology of their variables. Furthermore, the implemented routines should be of a reasonable size (preferably fewer than 80 lines of code), with limited fan–in and fan–out. Straightforward logic, natural expressions and conventional language should be followed [6,7].

The overall objective of maintainability is to improve some aspect of the software. Typical improvements are to restructure the code to reduce its complexity, improve its modularity (e.g., group the code implementing a feature into a single module), reduce the coupling between modules, minimize replicated code, and remove dead code. Other maintainability improvements relate to discovering and eliminating "anti-patterns" in the code, such as decision statements in which some values are not covered by branches, calls to external library functions which do not check the return value, or use of unsafe functions such as the C strcat routine [8].

Software maintainability is inversely proportional to both the effort required to make a change and the risk of breaking other functionality. The key targets in improving software maintainability are:

- Improve source code readability with comments and self-documented names
- Use a common programming language
- Keep software complexity low
- Loose coupling and high cohesion
- Isolate software functions using modularization techniques

Related Metrics:

- Comment Percentage in Code
- Modularity
- Number of Modules Score
- Module size score
- Cyclomatic Complexity
- Duplicate / Dead Code
- Number of Instances

TABLE III. MAINTAINABILITY SCORE MATRIX

Grade		Modules	*Size	Complexity (vG)	Duplicate/ unused Code	Languages
1	Very Good	>20	0-200	<3	< 10	<5
2	Good	15-20	201-300	3-6	11-25	5-7
3	Fair	11-15	301-400	7-10	26-40	7-9
4	Needs Improvement	5-10	401-500	11-14	41-50	10-12
5	Poor	1-5	>500	15-20	> 50	>12

The Maintainability score indicates how easy or hard it will be to upkeep the software. This score is determined mostly on software complexity, which is measured by identifying cyclomatic complexity [5]. Software with higher

complexity requires additional effort to upkeep or modify. This complexity is based on two attributes: (1) more effort is required to understand complex software and requires additional documentation as well as additional expertise and 2) additional effort is required to test because there are more independent paths require testing. Complex software is typically more prone to inherent defects, and repairing these defects can increase sustainment costs.

We assigned highest weighting of 50% to complexity since more complex code drives up the cost for testing and modifying the software [5]. Modularity provided additional 30% of the formula to account for code size and amount of modules that need to be maintained. The remaining 20% of the formula was equally distributed between programming languages and dead/duplicate code. Too many programming languages can increase upkeep cost. Duplicate and dead code introduce security risk factors as well as add to maintenance cost [5].

Calculation:

Modularity Calculation:

$Mo = (Mn \times 0.5) + (Ms \times 0.5)$

Complexity Calculation:

$Co = (Vg \times 0.5) + (Cp \times 0.5)$

Maintainability Calculation:

$M = Co(0.5) + Mo(0.3) + Dp(0.1) + Pl(0.1)$

M = Maintainability
Co = Complexity
Cp = Coupling
Vg = Cyclomatic Complexity
Pl = Programming Languages
Dp = Duplicate/Dead Code

D. Security

The Application Security and Development (ASD) Security Technical Implementation Guide (STIG) provides the baseline requirements for Government-Off-The-Shelf (GOTS) applications and may be used to evaluate custom-developed applications and Commercial-Off-The-Shelf (COTS) software. Static analysis tool output, such as Common Weakness Enumeration (CWE)/SANS Top 25 vulnerabilities may also be used to measure software security.

The Security Metric formula is based on multi-tier weighting. We implemented the multi-tier weighting to account for the disparate attributes associated with the security formula as well as the differing severity vulnerability rating scales provided by automated tool output. This formula assigns the highest value to Category I (CAT I) findings, followed by CATII and CATIII's and other potential issues within the system that may be elevated to CAT level in the future.

The project team defined the CAT formula to properly weight the associated severity of each classification of defects and assist in prioritizing vulnerabilities to be addressed. This formula does not however represent the application's overall risk . Risk assessment methodology in accordance with *DoDI 8510.01, Risk Management Framework (RMF) for DoD Information Technology (IT), NIST SP 800-30, Guide for Conducting Risk Assessments*, and Navy guidance should be used for risk management decisions.

Systems developed for Department of Navy use are typically are required to possess zero CATI findings in order to field. CATII findings can be present, but only with proper mitigation and a plan of action to mitigate or remediate those items during a defined time. CATIII findings are low risk and are allowed; however, every effort should be made to remedy these accordingly. Based on these criteria, each CAT finding classification is weighted as listed in the formula with CATI items weighted as (0.5) the total value, CATII weighted at (0.3) the total value, and CATIII weighted at (0.2) the total CAT value. Once the sum of these values is calculated, the CAT attribute is weighted for the Overall Security value.

Since Defect Density represents risk for potential issues, it makes up a significant attribute to define the overall security posture of a software application, it is imperative that it be given individual weighting and an attribute score in the Overall Security value. For the purpose of the formula, Defect Density was weighted at (0.25) of the Overall Security value. This weight is due to the fact that increasingly large numbers of software defects are found throughout the software we tested. This weight showed that the Defect Density could be considered very high. However, closer analysis revealed that the vast majority, approximately 80% of those defects, are trivial or minor in scope, and focused on coding style issues. These defects are believed to not impact the ability of the software to be secure and withstand cyber-related attack. Based on the premise that a large number of defects can be prevalent, it is not suggested that large numbers indicate proportionately large numbers of critical defects, but suggests the associated weighting of this attribute at the appropriate 0.25 score.

Related Metrics:

- Open Web Application Security Project (OWASP) Top 10 and CWE

TABLE IV. SECURITY SCORE MATRIX

	Security Scan Tool Vulnerabilities Observed		Source Code Scans	#Defects per KLOC	Priority	ASD STIG Map/CVSSv2 Qualitative Severity Rating Scale	
Grade		Severity	Severity %				
1	Very Good	None	<1%	<1		None	0
2	Good	Low / CAT III	1%-2.5%	2.5-1	4	Low / CAT III	0.1-3.9
3	Fair	Medium /Low	2.5%-4%	4-2.5	3	Medium / CAT II	4.0-6.9
4	Needs Improvement	Medium / CAT II	4%-10%	10-4	2	High / CAT I	7.0-8.9
5	Poor	Critical/High/CAT I	>10%	>10	1	Critical	9.0-10

Calculation:

Security Calculation:

S = (((CATI#s(.5) + CATII#s(.3) + CATIII#s(.2)) .75) + ((D / LOC) * .25)*

S = Security
LoC = Lines of Code
D =# Defects
CAT I = Any vulnerability, the exploitation of which will directly or immediately result in the loss of Confidentiality, Availability, or Integrity
CAT II = Any vulnerability, the exploitation of which has potential to result in loss of Confidentiality, Availability, or Integrity.
CAT III = Any vulnerability, the existence of which degrades measures to protect against loss of Confidentiality, Availability, or Integrity.

E. Extensibility

Extensibility can be confused with re-usability. Software extensibility describes how much effort is required to extend and change the software to provide new functionality that may not have been originally planned. Extensible design avoids software development issues such as low cohesion and high coupling.

Extensibility measures how easy or hard it will be to add to software's capability. Extensibility is impacted by various factors equally. These factors include software modularity, coupling/cohesion, complexity, and open architecture. Software that is modular can be easily extended because less code requires modification. Therefore, based on these values we equally weight the attributes for the Extensibility formula.

Related Metrics:

- Modularity
- Number of Modules Score
- Module Size Score
- Weighted Method per Class (WMC)
- Coupling / Cohesion
- Complexity
- Cyclomatic Complexity
- Architecture

TABLE V. EXTENSIBILITY SCORE MATRIX

Grade		Modules	*Size	WMC	Complexity	Cohesion (LOCM)	Coupling (CBO)	Architecture
1	Very Good	>20	0-200	1-2	<3	<90%	<2	1
2	Good	15-20	201-300	3-6	3-6	60-89%	2-4	2
3	Fair	11-15	301-400	<14	7-10	40-59%	4-6	3
4	Needs Improvement	5-10	401-500	14-20	11-14	25-39%	6-8	4
5	Poor	1-5	>500	>20	15-20	>25%	>8	5

We assigned equal weights of 33% to modularity, complexity and open architecture. We found each of these attributes to be equally important in determining extensibility. Our testing and empirical data showed that variation in weights of three factors had adverse effect on overall system extensibility [4,5].

Calculations:

Modularity Calculation:
Mo = (*Mn* x .5) + (*Ms* x .5)

Complexity Calculation:
Co = (*Vg* x .5) + (*Cp* x .5)

Extensibility Calculation:
E = *Mo*(.34) + *Co*(.33) + *Oa*(.33)

E = Extensibility
Mo = Modularity
Mn = Number of Modules
Ms = Module Size
Cp = Coupling
Co = Complexity
Vg = Cyclomatic Complexity
Oa = Open Architecture Assessment

F. Reliability

Software reliability is the measure of how well the software will work when a particular functionality is required. Issue density and software complexity are two key drivers impacting this metric. Software with fewer issues is less likely to breakdown when it executed. Software with lower complexity is typically easier to fix and test, requires less downtown time to fix any issues found, which improves availability, and in turn, increases reliability.

Our formula weighted the Software Issue Density and Cyclomatic Complexity values identical. Both contribute equally to the ability of a software application to maintain reliable operational use. While Cyclomatic Complexity produces the majority of the observed Software Issue Density due to risks associated with complex software, it also adds significant time in the repair of the associated defects encountered [5, 9].

Related Metrics:

- Software Issue Density
- Cyclomatic Complexity

TABLE VI RELIABILITY SCORE MATRIX

Grade		Bug Density in Modules	Cyclomatic Complexity
1	Very Good	<5%	<3
2	Good	5-10%	3-4
3	Fair	11-15%	7-10
4	Needs Improvement	16-20%	11-14
5	Poor	>20%	>15

Calculations:

Reliability Calculation:

$R = Dd(0.5) + Vg(0.5)$

R = Reliability
Dd = Defect Density
Vg = Cyclomatic Complexity

G. Testability

Testability is high when there is a high degree of controllability that enables the injection of any input combination and the invoking of any possible state or combination of state. To uncover faults, the ability to observe the state and behavior is another desirable characteristic. Complexity, modularity and size are also important factors in testability estimation.

Modular software was far easier to test because smaller, less complex modules that require less test paths through the source code to execute complete test coverage. We weighted the formula based on this fact. For this formula, we weighted Modularity (0.5), the established value of Testability.

A higher score in Testability also relies on a lower Cyclomatic Complexity value (0.4). A lower Cyclomatic Complexity score depicts less test paths necessary to exercise fully all branches through the software, which supports full test automation. This score is associated with the Modularity weighting, in that smaller modules, by necessity, would lend itself into having greater numbers, of less complex modules, and would then reduce the weighted complexity value. In addition, we determined the need to account for Dead/Unused code. This value accounts for a small variable (0.1), in the formula since higher complex modules were found to exhibit smaller amount of Dead or Unused Code in the tested modules or files being tested [4].

Related Metrics:

- Modularity
- Coupling/Cohesion
- WMC
- Number of Modules
- Module Size Score
- Cyclomatic Complexity
- Duplicate/Dead Code
- Number of Dead Code Instances

Calculation:
Modularity Calculation:

$Mo = (Mn \times .5) + (Ms \times .5)$

Testability Calculation:

$T = Mo(.5) + Vg(.4) + Dp(.1)$

T = Testability
Mo = Modularity
Mn = Number of Modules
Ms = Module Size
Vg = Cyclomatic Complexity
Dp = Duplicate/Dead Code

H. Scalability

The term "scalability" can encompass a wide range of meanings. For the purposes of this software quality model, scalability refers to how well the software performs given more users and data on a system representative of the production environment. The intent is to measure how the system adapts as the workload increases.

Without instrumenting the code or running performance tests, a quick measure of the scalability of the system depends on how well the software takes advantage of thread level and data level parallelism in addition to use of modular design. Software that exhibits more parallelism may have fewer dependencies and less coupling. Open architectures such as service oriented architectures (SOA) divide the system into composable parts that can adapt to varying demands.

Scalability should only be measured dynamically by monitoring resource utilization growth with increasing load. Depending on the intended platform, analysis tools such as Intel®, VTune™, Amplifier XE, HP® Loadrunner™, and Apache JMeter™, can be used to dynamically assess the software scalability.

I. Quality to Metrics Dependency Matrix

TABLE VII QUALITY TO METRICS DEPENDENCY MATRIX

	Modularity	Cyclomatic Complexity	Duplicate Code	Dead Code	Lines of Code/Comments	Coupling/Cohesion	Abstractness	Defect Density	OWASP Top 10	Weighted Method per Class (WMC)	Number of Classes	Size of Class	Number of Children per Class
Reusability	x	x				x	x			x	x	x	x
Portability	x					x					x	x	
Maintainability	x	x	x	x	x						x	x	
Security									x				
Extensibility	x	x				x	x			x	x	x	
Reliability		x						x					
Testability	x	x		x		x					x	x	
Scalability													

Modularity is separation of a software system in independent and collaborative modules that can be organized in a software architecture [10] . Modular software has several advantages such as maintainability, manageability, and comprehensibility.

There are five attributes closely related to modularity in software systems, size, coupling / dependency, complexity, cohesion, and information hiding. The first attribute is the size

of the module as well as the system that contains each module. It should not be too large in size. Additional features in the system should be translated as the addition in the module of the system. The second attribute is coupling / dependency which consist of direct / syntactic which can be achieved through composition, method signatures, class instantiations, and inheritance; and semantic or indirect coupling. The third attribute is complexity that can be measured by using software metrics such as McCabe's Cyclomatic Complexity or Halstead's Software Metrics. The fourth attribute is cohesion which measures the integrity of the code inside each of the module. The terms used to qualitatively measure cohesion are high cohesion or low cohesion. The last attribute is information hiding which involves hiding the details of implementation from external modules. Relating to the modularity property of a software system, in order to have an ideal modular software system, the software system should have the following attributes [11]:

- Small size in each module (package) and many modules in the system. Each module / package should only be responsible for a simple feature, and the more complex features should be composed of many of these simple features. The possible software metrics to measure size are NCLOC, Lines, or Statements.
- Low coupling / dependency: minimization or standardization of coupling / dependency e.g. through standard format i.e. published APIs, elimination of semantic dependencies, etc. The possible software metrics to measure coupling are Afferent Coupling, Efferent Coupling, or RFC (Response for a Class).
- Low complexity: hierarchy of modules that prefers flatter than taller dependency. The most popular software metrics to measure complexity is cyclomatic complexity. [12]
- High cohesion: high integrity of the internal structure of software modules which is usually stated as either high cohesion or low cohesion. The better measure of cohesion in object oriented programming such as Java™ is LCOM4 or Lack of Cohesion Metrics version 4 proposed by Hitz and Montazeri.
- Open for extension and close to modification: capability of the existing module to be extended to create a more complex module. And avoid changing already debugged code. The creation of new modules should be encouraged using available extension and not modifying the already tested module. [11]

B. Dependencies

Almost all software systems have components that are identifiable as data items, data types, subprograms, or source files. There is a dependency between two components if a change to one may have an impact that will require changes to the other.

C. Cyclomatic Complexity

The cyclomatic complexity of a section of source code is the number of linearly independent paths within it. For instance, if the source code contains no control flow statements (conditionals or decision points), such as IF statements, the complexity is 1, since there is only a single path through the code. If the code has one single-condition IF statement, there are two paths through the code: one where the IF statement evaluates to TRUE and another one where it evaluates to FALSE, so complexity is 2 for single IF statement with single condition. Two nested single-condition IFs, or one IF with two conditions, produces a complexity of 4, 2 for each branch within the outer conditional. Cyclomatic complexity was developed by Thomas J. McCabe, Sr. in 1976. [12]

One of McCabe's original applications was to limit the complexity of routines during program development; he recommended that programmers should count the complexity of the modules they are developing, and split them into smaller modules whenever the cyclomatic complexity of the module exceeds 10. This practice was adopted by the NIST Structured Testing methodology, with an observation that since McCabe's original publication, the figure of 10 has received substantial corroborating evidence, but that in some circumstances it may be appropriate to relax the restriction and permit modules with a complexity as high as 15. As the methodology acknowledged that there were occasional reasons for going beyond the agreed-upon limit, it phrased its recommendation as: "For each module, either limit cyclomatic complexity to [the agreed-upon limit] or provide a written explanation of why the limit was exceeded." [13]

D. Abstractness

Robert Martin proposed a widely used metric suite in 1994. Abstractness was included and is the ratio of the number of abstract classes versus the total number of classes. A value of 0 would mean no abstract classes were present and a max value of 1 would mean that all of the classes are abstract. [14]

Abstractness is important in order to maintain stability within the source code [15]. Abstractions allow the implementation to change without modifying the interfaces, so that dependent code does not break. Abstractions also may indicate the use of design patterns.

E. Coupling

This metric shows how strong the dependency of the source code is, i.e. the strength with which classes, methods, methods' parameters are connected to each other; the degree to which each program module relies on each one of the others.

Low (loose) coupling means that source code is organized in such a way, so that its methods and classes slightly address each other. That means that the source code is not written in the optimally; rather independent methods and classes were created to solve separate tasks.

F. Cohesion

This metric shows an average number of internal relationships per type in a package/namespace.

G. Afferent Coupling

Afferent means incoming. - This metric is applied to packages and namespaces. It is the number of types outside a package or namespace that depend on types of the current package or namespace. High afferent coupling shows that the analyzed package/ namespace has high importance.

H. Efferent Coupling

Efferent means outgoing. - It is the number of types inside a package/namespace that depend on types of other types/packages. High efferent coupling shows the degree to which the measured package/namespace depends on external packages/namespaces.

The main idea of this metric is that the class has high cohesion, when all its methods use all the fields of this class.

I. Duplicate Code

Code that is similar or copy and pasted can be harmful because it can increase maintenance costs and inconsistent changes to duplicate code can lead to inconsistent behavior. The presence of similar code also indicates the presence of a missed opportunity for reuse. [16]

J. Dead Code

Dead code is code that is never used. This includes unused methods and variables. Dead code can lead to difficulties in understanding the program which can lead to bugs or an increase in maintenance costs. [17]

K. Defect Density or Software Issue Density

Defect Density is the number of confirmed defects detected in software/component during a defined period of development/operation divided by the size of the software/component. [18]

Elaboration:

The 'defects' are:

- Confirmed and agreed upon (not just reported)
- Dropped defects are not counted

The period might be for one of the following:

- Duration (the first month, the quarter, or the year).
- For each phase of the software life cycle
- For the whole of the software life cycle

The size is measured in one of the following:

- Function Points (FP)
- Source Lines of Code

L. Weighted Methods per Class (WMC)

This metric is designed to provide a better measurement of class complexity. It is the sum of the complexities of all the class methods. A class having a high WMC is more complex and is harder to maintain, reuse or extend. Complexity is not explicitly defined for the metric to be generic. In the special case when complexity is not considered, the WMC metric is the same as the number of methods in the class.

M. Number of Children per class (NOC)

In Object Oriented (OO) terminology, classes that inherit their functionality from other classes are called Children Classes. A high value for NOC indicates that the class is implemented in abstract manner since other classes can inherit from it and reuse it.

VI. Static-Code Analysis Tools

Both industry and open-source developers have provided a wide array of useful static-code analysis tools.

A. Atomiq[19]

Summary: Atomiq is a free tool that finds duplicate and similar code.

Languages: C/C++, C#, Visual Basic.NET™, ASPX, Ruby™, Python™, Java™, ActionScript, XAML

Metrics Supported: Duplicate Code

B. Checkstyle[20]

Summary: Checkstyle is an open-source tool to help programmers write Java code that adheres to a coding standard. There is a plug-in for Eclipse™, IntelliJ IDEA™, Netbeans™, Jenkins™, and others that notifies developers on-the-fly of any violations.

Languages: Java™

Metrics Supported: Cyclomatic Complexity, Design For Extension, Presence of Javadoc Comments (packages, types, methods, variables), Magic Numbers, File Length, Method Length, Method Count

C. CLOC[21]

Summary: CLOC counts blank lines, comment lines, and physical lines of source code in many programming languages. Given two versions of a code base, CLOC can compute differences in blank, comment, and source lines. It is written entirely in Perl™ with no dependencies outside the standard distribution of Perl™ v5.6 and higher (code from some external modules is embedded within CLOC) and so is quite portable.

Languages:

Metrics Supported: Lines of Code, Lines of Comments, Lines of blank lines

D. CppDepend[22]

Summary: CppDepend simplifies managing a complex C/C++ code base. You can analyze code structure, specify design rules, do effective code reviews and master evolution by comparing different versions of the code. CppDepend counts the number of lines of code. It also

comes with more than 80 other code metrics. Some of them are related to your code organization (the number of classes or namespaces, the number of methods declared in a class...), some of them are related to code quality (complexity, percentage of comments, number of parameters, cohesion of classes, stability of Projects...), some of them are related to the structure of code (which types are the most used, depth of inheritance...)

Languages: C++

Metrics Supported: Similar to NDepend

E. FINDBUGS[23]

Summary: Open-source tool written by the University of Maryland used to find bugs in Java™ programs. A GUI is provided in addition to access via Ant.

Languages: Java™

Metrics Supported: Identifies code that follow common bug patterns for Java such as possible null pointer dereference or index out of bounds.

F. Find Security Bugs[24]

Summary: Open-source plugin for FindBugs providing security audits for Java™ web applications.

Languages: Java™

Metrics Supported: It can detect 63 different vulnerability types with over 200 unique signatures with extensive references given for each bug patterns with references to OWASP Top 10 and CWE.

G. FORTIFY™ [25]

Summary: Fortify™ by Hewlett Packard provides a comprehensive tool for detecting security vulnerabilities.

Languages: 21 languages

Metrics Supported: 500 types of vulnerability detection including OWASP Top 10

H. GMetrics[26]

Summary: The GMetrics project provides calculation and reporting of size and complexity metrics for Groovy source code. GMetrics scans Groovy source code, applying a set of metrics, and generates an HTML or XML report of the results.

Languages: Groovy

Metrics Supported: Cyclomatic Complexity, Afferent Coupling, Efferent Coupling, Lines per method, Lines per class, Number of classes per package, Number of field per class

I. JArchitect[27]

Summary: JArchitect offers a wide range of features. It is often described as a Swiss Army Knife for Java developers. JArchitect comes with more than 80 other code metrics. Some of them are related to your code organization (the number of classes or Packages, the number of methods declared in a class...), some of them are related to code quality (complexity, percentage of comments, number of parameters, cohesion of classes, stability of Projects...), some of them are related to the structure of code (which types are the most used, depth of inheritance...).

Languages: Java™

Metrics Supported: Similar to NDepend

J. McCabe IQ[27]

Summary: McCabe IQ provides software analysis tools to measure the complexity and quality of code at the application and enterprise level.

Languages: Ada, ASM86, C/C++, C#, C++.NET, COBOL, FORTRAN, Java™, JSP, Perl™, PL1, Visual Basic™,

Metrics Supported: Cyclomatic Complexity(<10), Module Design Complexity(<7), Essential Complexity (<4), Lack of Cohesion Methods (>75), Object Integration Complexity, Maintenance Severity

K. NDepend[28]

Summary: NDepend offers a wide range of features to let the user analyze a code base. It is often described as a Swiss Army Knife for .NET developers.

Languages: .NET™

Metrics Supported: Lines of code, Lines of comments, Afferent coupling, Efferent coupling, Abstractness, Instability, Lack of cohesion of methods, Cyclomatic Complexity (<10)

L. PMD™ [29]

Summary: PMD™ is an open-source tool used to find defects, including possible bugs,
dead code, suboptimal code, overcomplicated expressions, and duplicate code.

Languages: PMD™ supports rulesets for Java™, Javascript™, JSP, PL/SQL, Velocity Template Language, and XML/XSL. The PMD™ Copy Paste Detector can run with additional languages including C++, C#, Fortran, Go, Matlab™, etc.

Metrics Supported: Varies by language. For most languages, copy paste detection is provided. For Java™, additional metrics include: Source lines of code, Cyclomatic Complexity (<10), Coupling Between Objects, Loose Coupling, Exception Handling, Unused Code (Dead Code)

M. *SonarQube™ [30]*

Summary: SonarQube™ is an open-source platform for managing code quality. The tool supports 20+ languages through plug-ins and can collect a variety of metrics in addition to allowing the creation of custom metric rules. It also supports a variety of plug-ins for other code analysis tools such as Checkstyle and PMD™ that can extend the number of metrics it can collect.

Languages: Java™, C#, C/C++, PL/SQL, Cobol, ABAP™, …(20+ languages supported through plug-ins)

Metrics Supported: Duplicate Code, Failed unit tests, Insufficient branch coverage by unit tests, Insufficient comment density, Insufficient line coverage by unit tests, skipped unit tests

N. *UCC[31]*

Summary: UCC is a comprehensive source lines of code counter produced by the
USC Center for Systems and Software Engineering. It is an open-source tool that can be compiled with any ANSI™ standard C++ compiler.

Languages: C/C++, C#, Java™, Visual Basic™, Assembly, and others

Metrics Supported: SLOC, PSLOC, LSLOC

O. *UNDERSTAND[32]*

Summary: Understand is a robust static code analysis tool developed by Scientific Toolworks, Inc. supporting the generation of multiple kinds of reports and views of the data at different levels (project, class, object oriented metrics, program unit, file). Understand can perform dependency analysis in addition to code standards testing.

Languages: Ada, COBOL, Coldfire 68K Assembly, C/C++, C#, Fortran, Java™, Jovial, Pascal™, PL/M, Python™, VHDL, Javascript™, PHP™, XML, HTML, CSS

Metrics Supported: Understand can check for adherence to published coding standards from Effective C++ (3rd Edition) by Scott Meyers, MISRA-C 2004, MISRA-C++ 2008, and any custom coding standards defined by the user. Understand also supports checks for Dead Code, Cyclomatic Complexity, SLOC, Coupling Between Objects, Lack of Cohesion in Methods, comment to code ratio.

P. *Tools to Metrics Matrix*

TABLE VIII. TOOLS TO METRICS MATRIX

	Cyclomatic Complexity	Duplicate Code	Dead Code	Lines of Code/Comments	Counling/Cohesion	Abstractness	Defect Density	OWASP Top 10	Number of Classes	Size of Class	Number of Children per Class
Atomiq		x		x							
Checkstyle	x										
CLOC				x							
CppDepend	x			x	x	x				x	x
FINDBUGS							x				
Find Security Bugs							x	x			
Fortify							x	x			
GMetrics	x			x	x				x	x	
JArchitect	x			x	x	x				x	x
McCabe IQ	x										
NDepend	x			x	x	x				x	x
PMD		x	x		x		x				
SonarQube (no plug-ins)		x		x							
UCC				x							
UNDERSTAND	x	x	x	x	x		x				

CONCLUSION

We have identified software qualities, software analysis tools, and related metrics. This effort was based on existing data and analysis of that data, proofing a formula for use by Department of the Navy software development efforts to measure inherent Quality of the software being developed. However, each project is unique and will require a software quality model tailored for its individual needs.

This process is an ongoing effort for any organization and requires analysis of data and trends to determine the most effective implementation of metrics to achieve the highest fidelity of quality and provide for beneficial cost savings. In addition, evaluators need to create and calibrate cost functions for the cost of fixing the code that does not meet software code requirements. This allows for the normalization of the software model based on cost.

REFERENCES

[1] [1] J. S. Poulin, “Measuring Software Reusability”; Third International Conference on Advances in Software Reusability, 1994.

[2] S. Chidamber, C.Kemerer, “A Metrics Suite for Object Oriented Design”, IEEE Transactions of Software Engineering, Vol.20, NO.6, June 1994.

[3] J. Lenhard, G. Wirtz, “Measuring the Portability of Executable Service-Oriented Processes”; Distributed Systems Group, University of Bamberg, Bamberg, Germany.

[4] J. D. Mooney, “Issues in the Specification and Measurement of Software Portability”; Dept. of Statistics and Computer Science, West Virginia, PO BOX 6330, Morgantown, WV. 26506-6330

[5] L. Rosenberg PhD, T. Hammer, J. Shaw, “Software Metrics and Reliability”, GSFC, UNISYS/NASA.

[6] Kernighan, B, and Pike, P, ‘*The Practice of Programming*’, Addison Wesley, isbn: 0-201-61586-X, 1999.

[7] D. Stabrinoudis, M. Xenos, D. Christodoulakis, “Relation Between Software Metrics and Maintainability”; Technical Report No. TR99/08/03, Computer Technology Institute, 1999

[8] F. Weil, PhD., “Modernized and Maintainable Code”, UniqueSoft, LLC. 2015

[9] M. Lacchia, “ Introduction to Code Metrics”, http://radon.readthedocs.org/en/latest/intro.html

[10] d. S. E. d. B. M. K. Nakagawa E.Y, "Software Architecture Relevance in Open Source Software Evolution: A Case Study," in Annual IEEE International Computer Software and Application Conference, 2008.

[11] R. W. J. E. I. K. M. Andi Wahju Rahardjo Emanuel, "STATISTICAL ANALYSIS ON SOFTWARE METRICS AFFECTING MODULARITY IN OPEN SOURCE SOFTWARE," International Journal of Computer Science & Information Technology (IJCSIT), vol. 3, pp. 105-118, 2011.

[12] T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, Vols. SE-2, no. 4, 1976.

[13] A. H. Watson and T. J. McCabe, "NIST Special Publication 500-235 Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," McCabe Software, 1996.

[14] Y. Suresh, J. Pati and S. K. Rath, "Effectiveness of Software Metrics for Object-oriented System," Procedia Technology, vol. 6, pp. 420-427, 2012.

[15] R. Martin, "Object Mentor," 28 October 1994. [Online]. Available: http://www.objectmentor.com/resources/articles/oodmetrc.pdf. [Accessed 15 August 2015].

[16] E. Juergens, F. Deissenboeck and B. Hummel, "Code Similarities Beyond Copy & Paste," Proceedings of the 14th European Conference on Software Maintenance and Reengineering, 2010.

[17] "Detect Dead Code and Calls to Deprecated Methods with Sonar Squid," SonarQube, [Online]. Available: http://www.sonarqube.org/detect-dead-code-and-calls-to-deprecated-methods-with-sonar-squid/. [Accessed 14 August 2015].

[18] softwaretestingfundamentals, "softwaretestingfundamentals.com," [Online]. Available: http://softwaretestingfundamentals.com/defect-density/.

[19] "Atomiq Code Similarity Finder," [Online]. Available: http://www.getatomiq.com/. [Accessed 31 July 2015].

[20] "CheckstyleChecks," [Online]. Available: http://checkstyle.sourceforge.net/checks.html. [Accessed 31 July 2015].

[21] Codergears, "CppDepend," Codergears, [Online]. Available: http://cppdepend.com. [Accessed 14 August 2015].

[22] "FindBugs Bug Descriptions," [Online]. Available: http://findbugs.sourceforge.net/bugDescriptions.html. [Accessed 31 July 2015].

[23] P. Arteau, Open-source, open for contributions, [Online]. Available: http://h3xstream.github.io/find-sec-bugs.

[24] H. Packard, "Securing your enterprise software," [Online]. Available: http://www8.hp.com/us/en/software-solutions/asset/software-asset-viewer.html?asset=1356157&module=1823975&docname=4AA4-2455ENW&page=1823980. [Accessed 29 August 2015].

[25] "Gmetrics," Apache License V2.0, [Online]. Available: http://gmetrics.sourceforge.net/index.html.

[26] Codergears, "jarchitect," Codergears, [Online]. Available: http://www.jarchitect.com.

[27] "McCabe Software Metrics Glossary," McCabe Software, [Online]. Available: http://www.mccabe.com/iq_research_metrics.htm. [Accessed 31 July 2015].

[28] Codergears, "ndepend," Codergears, [Online]. Available: http://www.ndepend.com.

[29] "PMD Rulesets index: Current Rulesets," [Online]. Available: https://pmd.github.io/pmd-5.3.3/pmd-java/rules/index.html. [Accessed 31 July 2015].

[30] "SonarQube Manual Issues," SonarSource S.A., [Online]. Available: http://docs.sonarqube.org/display/SONAR/Manual+Issues. [Accessed 3 August 2015].

[31] "UCC About," USC Center for Systems and Software Engineering, [Online]. Available: http://csse.usc.edu/ucc_wp/about/. [Accessed 31 July 2015].

[32] "Understand Features," Scientific Toolworks, Inc., [Online]. Available: https://scitools.com/features. [Accessed 3 August 2015].